Explicit Control in a Batch-Aware Distributed File System

John Bent, Douglas Thain,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny
Computer Sciences Department, University of Wisconsin, Madison

Abstract

We present the design, implementation, and evaluation of the Batch-Aware Distributed File System (BAD-FS), a system designed to orchestrate large, I/O-intensive batch workloads on remote computing clusters distributed across the wide area. BAD-FS consists of two novel components: a storage layer that exposes control of traditionally fixed policies such as caching, consistency, and replication; and a scheduler that exploits this control as necessary for different workloads. By extracting control from the storage layer and placing it within an external scheduler, BAD-FS manages both storage and computation in a coordinated way while gracefully dealing with cache consistency, fault-tolerance, and space management issues in a workload-specific manner. Using both microbenchmarks and real workloads, we demonstrate the performance benefits of explicit control, delivering excellent end-to-end performance across the wide-area.

1 Introduction

Traditional distributed file systems, such as NFS and AFS, are built on the solid foundation of empirical measurement. By studying expected workload patterns [7, 41, 45, 50, 57], researchers and developers have long been able to make appropriate trade-offs in system design, thereby building systems that work well for the workloads of interest. Most previous distributed file systems have been targeted at a particular computing environment, namely a collection of interactively used client machines. However, as past work has demonstrated, different workloads lead to different designs (*e.g.*, FileNet [18] and the Google File System [29]); if assumptions about usage patterns, sharing characteristics, or other aspects of the workload change, one must reexamine the design decisions embedded within distributed file systems.

One area of increasing interest is that of batch workloads. Long popular within the scientific community, batch computing is increasingly common across a broad range of important and often commercially viable application domains, including genomics [3], video production [52], simulation [11], document processing [18], data mining [2], electronic design automation [17], financial services [42], and graphics rendering [36].

Batch workloads minimally present the system with a set of jobs that need to be run and perhaps some ordering among them; in many environments, the approximate run times and I/O requirements are also known in advance. A scheduler uses this information to dispatch jobs so as to maximize throughput.

Batch workloads are typically run in controlled local-area cluster environments. However, organizations that have large workload demands increasingly need ways to share resources across the wide-area, both to lower costs and to increase productivity. One approach to accessing resources across the wide-area is to simply run a local-area batch system across multiple clusters that are spread over the wide-area and to use a distributed file system as a backplane for data access.

Unfortunately, this approach is fraught with difficulty, largely due to the way in which I/O is handled. The primary problem with using a traditional distributed file system is in its approach to *control*: many decisions concerning caching, consistency, and fault tolerance are made *implicitly* within the file system. Although these decisions are reasonable for the workloads for which these file systems were designed, they are ill-suited for a wide-area batch computing system. For example, to minimize data movement across the wide-area, the system must carefully use the cache space of remote clusters; however, caching decisions are buried deep within distributed file systems, thus preventing such control.

To mitigate these problems and enable the utilization of remote clusters for I/O-intensive batch workloads, we introduce the Batch-Aware Distributed File System (BAD-FS). BAD-FS differs from traditional distributed file systems in its approach to control; BAD-FS exposes decisions commonly hidden inside of a distributed file system to an external workload-savvy scheduler. BAD-FS leaves all consistency, caching, and replication decisions to this scheduler, thus enabling *explicit* and workload-specific control of file system behavior.

The main reason to migrate control from the file system to the scheduler is *information* – the scheduler has intimate knowledge of the workload that is running and can exploit that knowledge to improve performance and streamline failure handling. The combination of workload information and explicit control of the file system leads to three distinct benefits over traditional approaches:

• **Enhanced performance.** By carefully managing remote cluster disk caches in a cooperative fashion, and by controlling I/O such that only needed data is transported across the wide-area, BAD-FS minimizes wide-area traffic and improves throughput. Using workload knowledge, BAD-FS further improves performance by using capacity-aware scheduling to avoid thrashing.

• **Improved failure handling.** Using detailed workload information, the scheduler can determine whether to make replicas of data based on the cost of generating it, and not indiscriminately as is typical in many file systems. Data loss is treated uniformly as a performance problem. The scheduler has the ability to regenerate a lost file by rerunning the application that generated it and hence only replicates when the cost of regeneration is high.

• **Simplified implementation.** Detailed workload information allows a simpler implementation. For example, BAD-FS provides a cooperative cache but does not implement a cache consistency protocol. Through exact knowledge of data dependencies, it is the scheduler that ensures proper access ordering among jobs. Previous work has demonstrated the difficulties of building a more general cooperative caching scheme [4, 12].

We demonstrate the benefits of explicit control via our prototype implementation of BAD-FS. Using synthetic workloads, we demonstrate that BAD-FS can reduce wide-area I/O traffic by an order of magnitude, can avoid performance faults through capacity-aware scheduling, and can proactively replicate data to obtain high performance in spite of remote failure. Using real workloads, we demonstrate the practical benefits of our system: I/O-intensive batch workloads can be run upon remote resources both easily and with high performance.

Finally, BAD-FS achieves these ends while maintaining site autonomy and support for unmodified legacy applications. Both of these practical constraints are important for acceptance in wide-area batch computing environments.

The rest of this paper is organized as follows. In Section 2, we describe our assumptions about the expected environment and workload, in Section 3, we discuss the architecture of our system, in Section 4, we present our experimental evaluation, in Section 5, we examine related work, and finally in Section 6, we conclude.

2 Background

In this section, we describe the setting for BAD-FS. We present the expected workloads, basing assumptions on our recent work in batch workload characterization [54]. We describe the computing environment available to users of these workloads and the difficulty they encounter executing such workloads with conventional tools.

2.1 Workloads

As illustrated in Figure 1, these data-intensive workloads are composed of multiple independent vertical sequences of processes that communicate with their ancestors and relatives via private data files. A workload generally consists of a large number of these sequences that are incidentally synchronized at the beginning, but are logically distinct and may correctly execute at a different rate than their siblings. We refer to a vertical slice of the workload as a *pipeline*, a horizontal slice as a *batch*, and the entire set as a *batch-pipelined* workload. Note that "pipeline" is used generically; the processes are *not* connected by UNIX-style pipes but rather communicate through files.

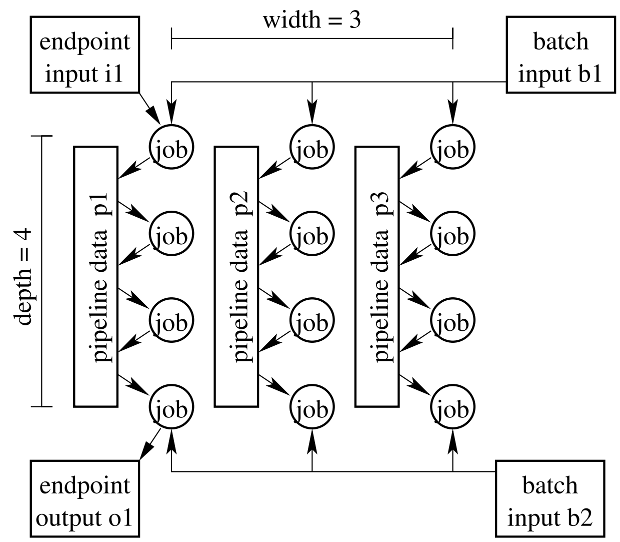


Figure 1: **A Typical Batch-Pipelined Workload.** *A single pipeline represents the logical work that a user wishes to complete, and is comprised of a series of jobs. Users often assemble many such pipelines into a batch to explore variations on input parameters or other input data.*

One of the key differences between a single application and a batch-pipelined workload is file sharing behavior. For example, when many instances of the same pipeline are run, the same executable and potentially many of the same input files are used. We characterize the sharing that occurs in these batch-pipelined workloads by breaking I/O activity into three types (as shown in Figure 1): *endpoint*, the unique input and final output; *pipeline-shared*, shared write-then-read data within a single pipeline; and *batch-shared*, input data shared across multiple pipelines.

2.2 Environment

Although wide-area sharing of untrusted and arbitrary personal computers is a possible platform for batch workloads [53], we believe that a better platform for these types of throughput-intensive workloads is one or more clusters of managed machines, spread across the wide area. We assume that each cluster machine has processing, memory, and local disk space available for remote users, and that each cluster exports its resources via a CPU sharing system. The obvious bottleneck of such a system is the wide-area connection, which must be managed carefully to ensure high performance. For simplicity, we focus most of our efforts on the case of a single cluster being accessed by a remote user. However, in Section 4.8, we present preliminary results from a multi-cluster environment.

We refer to this more organized, less hostile, and well managed collection of clusters as a *c2c (cluster-to-cluster)* system, in contrast to popular peer-to-peer (p2p) systems. Although the p2p environment is appropriate for many uses, there is likely to be a more organized effort to share computing resources within corporations or other organizations. We may assume that c2c environments are more stable, more powerful, and more trustworthy. That said, p2p technologies and designs are likely to be directly applicable to the c2c domain.

We also make the practical and important assumption that each site has local autonomy over its resources. Autonomy has two primary implications on the design of BAD-FS. First, although a workload may be able to use remote resources at a given time, these resources may be arbitrarily revoked. Thus, a system that is built to exploit remote resources must be able to tolerate unexpected resource failures, whether they are due to physical breakdowns, software failures, or deliberate preemptions. Second, autonomy prohibits the deployment of arbitrary software within the remote cluster. In designing BAD-FS, we assume that a remote cluster only provides us with the ability to dispatch a well-defined job as an ordinary, unprivileged user. Mandating that a single distributed file system be used everywhere is not a viable solution.

Finally, we assume that the jobs run on these systems cannot be modified, In our experience, many scientific workloads are the product of years of fine-tuning, and when complete, are viewed as untouchable. Also, ease of use is important; the less work for the user, the better.

2.3 Current Solutions

We now consider a user who wishes to run a batch-pipelined workload in this environment. After the user has developed and debugged the workload on their home system, they are ready to run batches of hundreds or thousands on all available computing resources, using remote batch execution systems such as Condor, LSF, PBS, or Grid Engine. Each pipeline in their workload is expected to use much of the same input data, while varying parameters and other small inputs. The necessary input data begins on the user's *home storage server* (*e.g.*, an FTP server), and the output data, when generated, should eventually be committed to this home server. Conventional batch computing systems present a user with two options for running a workload.

The first option, *remote I/O*, is to simply submit the workload to the remote batch system. With this option, all input and output occur on demand back to the home storage device. Although this approach is simple, the throughput of a data-intensive workload will be drastically reduced by two factors. First, wide-area network bandwidth is not sufficient to handle simultaneous batch reads from many data-intensive pipelines running in parallel. Second, all pipeline output is directed back to the home site, including temporary data that is not needed after the computation completes.

The second option, *pre-staging*, is for the user to manually configure the system to replicate batch data sets in the remote environment. This approach requires the user to have or obtain an account in the remote environment, identify the necessary input data, transfer that data to the remote site, log into the remote system, unpack the data in an appropriate location, configure the workload to recognize the correct directories (possibly using `/tmp` for temporary pipeline data), submit the workload, and manually deal with any failures. The entire process must be repeated whenever more data needs to be processed, new batch systems become available, or existing systems no longer have capacity to offer to the user. As is obvious from the description, this configuration process is labor-intensive and error-prone; additionally, using `/tmp` can be challenging because its availability cannot be guaranteed. Another limitation is that because the user has made these configurations independently of the scheduling system, the scheduling system is not able to correctly checkpoint pipelines within the workload. Still, many users go to these lengths simply to run their workloads.

Traditional distributed file systems would be a better solution but are typically not available due to administrative desire to preserve autonomy across domain boundaries. Even were such systems available, their fixed policies prevent them from being viable for batch-pipelined workloads. Consider, for example, BLAST [3], a commonly used genomic search program, consisting of a single stage pipeline that searches through a large shared dataset for protein string matches. Assume a user were to run BLAST on a compute cluster of 100 nodes equipped with a conventional distributed file system such as AFS or NFS. With cold caches, all 100 nodes will individually (and likely simultaneously) access the home server with the same large demands, resulting in poor performance as the dataset is redundantly transferred over the wide area network. Once the caches are loaded, each node will run at local disk speeds, but only if the dataset can fit in its cache. If it cannot, the node will thrash and generate an enormous amount of repetitive traffic back to the home server. Further, lacking workload information, each node must employ some mechanism to protect the consistency and availability of its cached data.

In contrast, a batch-aware system such as BAD-FS has a global view of the hardware configuration and workflow structure; it can execute such workloads much more efficiently by copying the dataset a single time over the wide

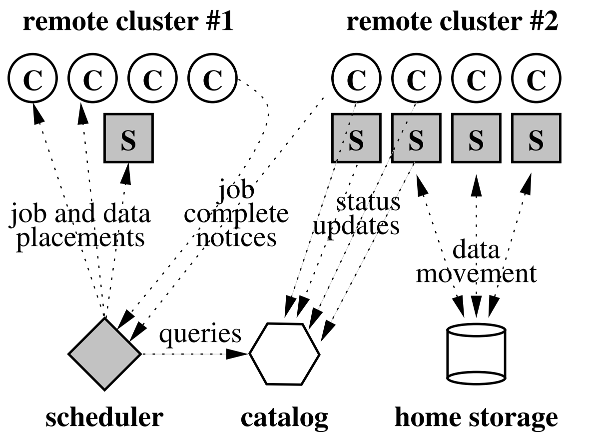


Figure 2: **System Architecture.** *Circles are compute servers, which execute batch jobs. Squares are storage servers, which hold cached inputs and temporary outputs. Both types of servers report to a catalog server, which records the state of the system. The scheduler uses information from the catalog to direct the system by configuring storage devices and submitting batch jobs. The gray shapes are novel elements in our design; the white are standard components found in batch systems.*

area and sharing or duplicating the data at the remote cluster. Further, explicit knowledge of sharing characteristics permits such a system to dispense with the expense and complexity of consistency checks while allowing nodes to continue executing even while disconnected.

3 Architecture

In this section, we present the architecture and implementation of BAD-FS. Recall that the main goal of the design of BAD-FS is to export sufficient control to a remote scheduler to allow it to deliver improved performance and better fault-handling for I/O-intensive batch workloads run on remote clusters. Figure 2 summarizes the architecture of BAD-FS, with the novel elements shaded gray.

BAD-FS is structured as follows. Two types of server processes manage local resources. A *compute server* exports the ability to transfer and execute an ordinary user program on a remote CPU. A *storage server* exports access to disk and memory resources via remote procedure calls that resemble standard file system operations. It also permits remote users to allocate space via an abstraction called *volumes*. *Interposition agents* bind unmodified workloads running on compute servers to storage servers. Both types of servers periodically report themselves to a *catalog server*, which summarizes the current state of the system. A *scheduler* periodically examines the state of the catalog, considers the work to be done, and assigns jobs to compute servers and data to storage servers. The scheduler may obtain data, executables, and inputs from any number of external storage sites. For simplicity, we assume the user has all the necessary data stored at a single *home storage server* such as a standard FTP server.

From the perspective of the scheduler, compute and storage servers are logically independent. A specialized device might run only one type of server process: for example, a diskless workstation runs only a compute server, whereas a storage appliance runs only a storage server. However, a typical workstation or cluster node has both computing and disk resources and thus runs both.

BAD-FS may be run in an environment with multiple owners and a high failure rate. In addition to the usual network and system errors, BAD-FS must be prepared to for *eviction* failures in which shared resources may be revoked without warning. The rapid rate of change in such systems creates possibly stale information in the catalog. BAD-FS must also be prepared to discover that the servers it attempts to harness may no longer be available.

BAD-FS makes use of several standard components. Namely, the compute servers are Condor [38] *startd* processes, the storage servers are modified NeST storage appliances [8], the interposition agents are Parrot [55] agents, and the catalog is the Condor *matchmaker*. The servers advertise themselves to the catalog via the ClassAd [43] resource description language.

3.1 Storage Servers

Storage servers are responsible for exporting the raw storage of the remote sites in a manner that allows efficient management by remote schedulers. A storage server does not have a fixed policy for managing its space. Rather, it makes several policies accessible to external users who may carve up the available space for caching, buffering, or other tasks as they see fit. Using an abstraction called *volumes*, storage servers allow users to allocate space with a name, a lifetime, and a type that specifies the policy by which to internally manage the space. The BAD-FS storage server exports two distinct volume types: scratch volumes and cache volumes.

A *scratch volume* is a self-contained read-write file system, typically used to localize access to temporary data. The scheduler can use scratch volumes for pipeline data passed between jobs and as a buffer for endpoint output. Using scratch volumes, the scheduler minimizes home server traffic by localizing pipeline I/O and only writing endpoint data when a pipeline successfully completes.

A *cache volume* is a read-only view of a home server, created by specifying the name of the home server and path, a caching policy (*i.e.*, LRU or MRU), and a maximum storage size. Multiple cache volumes can be bound into a *cooperative cache volume* by specifying the name of a catalog server, which the storage servers query to discover their peers. A number of algorithms [16, 20] exist for managing a cooperative cache, but it is not our intent to explore the range of these algorithms here. Rather, we describe a reasonable algorithm for this system and explain how it is used by the scheduler.

The cooperative cache is built using a distributed hash table [31, 37]. The keys in the table are block addresses, and the values specify which server is primarily responsible for that block. To avoid wide-area traffic, only the primary server will fetch a block from the home server and the other servers will create secondary copies from the primary. When space is needed, secondary data is evicted before primary. To approximate locality, our initial implementation only forms cooperative caches between peers in the same subnetwork. Although our initial analysis suggests that this is sufficient, in the future we plan on investigating other more complicated grouping algorithms.

Failures within the cooperative cache, including partitions, are easily managed but may cause slowdown. Should a cooperative cache be internally partitioned, the primary blocks that were assigned to the now missing peers will be reassigned. As long as the home server is accessible, partitioned cooperative caches will be able to refetch any lost data and continue without any noticeable disturbance to running jobs.

This approach to cooperative caching has two important differences from previous work. First, because data dependencies are completely specified by the scheduler, we do not need to implement a cache consistency scheme. Once read, all data are considered current until the scheduler invalidates the volume. This design decision greatly simplifies our implementation; previous work has demonstrated the many difficulties of building a more general cooperative caching scheme [4, 12]. Second, unlike previous cooperative caching schemes that manage cluster memory [16, 20], our cooperative cache stores data on local *disks*. Although managing memory caches cooperatively could also be advantageous, the most important optimization to make in our environment is to avoid data movement across the wide-area; managing remote disk caches is the simplest and most effective way to do so.

3.1.1 Local vs. Global Control. Note that volumes export only a certain degree of control to the scheduler. Namely, by creating and deleting volumes, the scheduler controls which data sets reside in the remote cluster. However, the storage servers retain control over per-block decisions. Two such important decisions made locally by the storage servers are the assignment of primary blocks in the cooperative cache and cache victim selection. Of course, if the scheduler is careful in space allocation, the cache will only victimize blocks that are no longer needed. In general, we have found this separation of global and local control to be suitable for our workloads. Although more work needs to be done to precisely identify the balance point, it is clear that a trade-off is better than either extreme. Complete local control, the current approach, suffers because the policies embedded within distributed file systems are inappropriate for batch workloads. The other extreme, complete global control, in which the scheduler makes decisions for each block of data, would require exorbitant complexity in the scheduler and would incur excessive network traffic to exert this fine-grained control.

3.2 Interposition Agents

In order to permit ordinary workloads to make use of storage servers, an *interposition agent* [33] transforms POSIX I/O operations into storage server calls. The agent's mapping from logical path names to physical storage volumes is provided by the scheduler at runtime. Together, the agent and the volume abstraction can hide a large number of errors from the job and the end user. For example, if a volume no longer exists, whether due to accidental failure or deliberate preemption, a storage server returns a unique *volume lost* error to the agent. Upon discovering this, the agent forcibly terminates the job, indicating that it could not run correctly in the given environment. This gives the scheduler clear indication of failures and allows it to take transparent recovery actions.

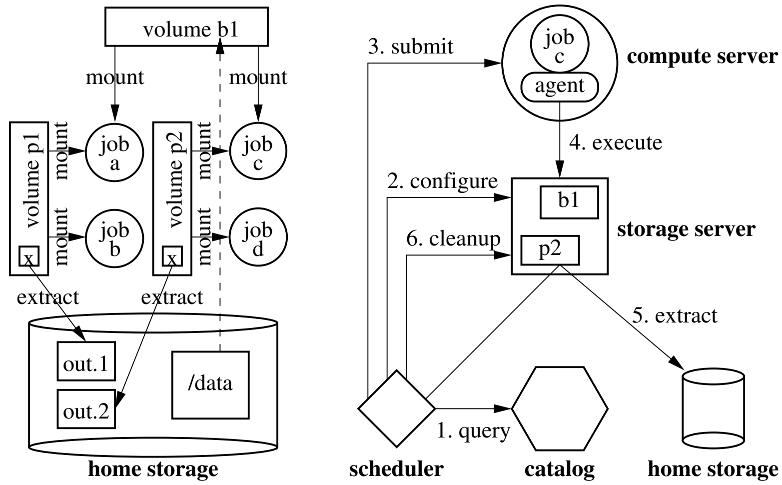
3.3 The Scheduler

The BAD-FS scheduler directs the execution of a workload on compute and storage servers by combining a static workload description with dynamic knowledge of the system state. Specifically, the scheduler minimizes traffic across the wide-area by differentiating I/O types and treating each appropriately, carefully managing remote storage to avoid thrashing and replicating output data proactively if that data is expensive to regenerate.

3.3.1 Workflow language. Shown in Figure 3 is an example of the declarative workflow language that describes a batch-pipelined workload and shows how the scheduler converts this description into an execution plan. The keyword `job` names a job and binds it to a description file, which specifies the information needed to execute that job. `Parent` indicates an ordering between two jobs. The `volume` keyword names the data sources required by the workload. For example, volume `b1` comes from an FTP server, while volumes `p1` and `p2` are empty scratch volumes. Volume sizes are provided to allow the scheduler to allocate space appropriately. The `mount` keyword binds a volume into a job's namespace. For example, jobs `a` and `c` access volume `b1` as `/mydata`, while jobs `a` and `b` share volume `p1` via the path `/tmp`. The `extract` command indicates which files of interest must be committed to the home server. In this case, each pipeline produces a file `x` that must be retrieved and uniquely renamed.

To many readers accustomed to working in an interactive environment, this language may seem like an unusual burden. We point out that a user intending to execute batch-pipelined workloads must be exceptionally organized. Batch users already provide this information, but it is scattered across shell scripts, make files, and batch submission files. In addition to imparting needed information to the BAD-FS scheduler, this workflow language actually reduces user burden by collecting all of this dispersed information into a coherent whole.

```
job      a    a.condor
job      b    b.condor
job      c    c.condor
job      d    d.condor
parent   a    child    b
parent   c    child    d
volume   b1   ftp://home/data 1 GB
volume   p1   scratch 50 MB
volume   p2   scratch 50 MB
mount    b1   a    /mydata
mount    b1   c    /mydata
mount    p1   a    /tmp
mount    p1   b    /tmp
mount    p2   c    /tmp
mount    p2   d    /tmp
extract  p1   x    ftp://home/out.1
extract  p2   x    ftp://home/out.2
```



(A) (B) (C)

Figure 3: **Workflow and Scheduler Examples.** (A) *A simple workflow script. A directed graph of jobs is constructed using* job *and* parent*, and the file system namespace presented to jobs is configured with* volume *and* mount*. The* extract *keyword indicates which files must be committed to the home storage server after pipeline completion.* (B) *A graphical representation of this workflow.* (C) *The scheduler's plan for job c. (1) The scheduler queries the catalog for the current system state and decides where to place job c and its data. (2) The scheduler creates volumes b1 and p2 on a storage server. (3) Job c is dispatched to the compute server. (4) Job c executes, accessing its volumes via the agent. (5) After jobs c and d complete, the scheduler extracts x from p2. (6) The scheduler frees volumes b1 and p2.*

3.3.2 I/O Scoping. Unlike most file systems, BAD-FS is aware of the flow of its data. From the workflow language, the scheduler knows where data originates and where it will be needed. This allows it to create a customized environment for each job and minimize traffic to the home server. We refer to this as *I/O scoping*.

I/O scoping minimizes traffic in two ways. First, cooperative cache volumes are used to hold read-only batch data such as `b1` in Figure 3. Such volumes may be reused without modification by a large number of jobs. Second, scratch volumes, such as `p2` in Figure 3, are used to localize pipeline data. As a job executes, it accesses only those volumes that were explicitly created for it; the home server is accessed only once for batch data and not at all for pipeline.

3.3.3 Consistency management. With the workload information expressed in the workflow language, the scheduler neatly addresses the issue of consistency management. All of the required dependencies between jobs and data are specified directly. Since the scheduler only runs jobs so as to meet these constraints, there is no need to implement a cache consistency protocol among the BAD-FS storage servers.

The user may make mistakes in the workflow description that can affect both cache consistency and correct failure recovery. However, through an understanding of the expected workload behavior as specified by the user, the scheduler can easily detect these mistakes and warn the user that the results of the workload may have been compromised. We have not yet implemented these detection features, but the architecture readily admits them.

3.3.4 Capacity-Aware Scheduling. The scheduler is responsible for throttling a running workload to avoid performance faults and maximize throughput. By carefully allocating volumes, the scheduler avoids overflowing storage or thrashing caches. Although disk capacity is rapidly increasing, the size of data sets is also growing and space management remains important [4, 6, 23]. The scheduler manages space by retrieving a list of available storage from the catalog server and selecting the ready job with the least unfulfilled storage needs, whether pipe or batch. If the scheduler is able to allocate all of that job's volumes, then it allocates and configures these volumes and schedules the job. If there are no jobs to execute or not enough available space, then the scheduler waits for a job to complete, more resources to arrive, or for a failure to occur. Note that due to a lack of complete global control, the scheduler may need to slightly overprovision when the needed volume size approaches the storage capacity.

In other scheduling domains, selecting the smallest job first can result in starvation. In this domain, however, starvation is avoided because a workflow is a static entity executed by one scheduler. Although smaller jobs will run first, all jobs will eventually be run.

3.3.5 Failure Handling. Finally, the scheduler makes BAD-FS robust to failures by handling failures of jobs, storage servers, the catalog, and itself. One aspect of batch workloads that we leverage is job idempotency; a job can simply be rerun in order to regenerate its output.

The scheduler keeps a log of allocations in persistent storage, and uses a transactional interface to the compute and storage servers. If the scheduler fails, then allocated volumes and running jobs will continue to operate unaided. If the scheduler recovers, it simply re-reads the log to discover what resources have been allocated and resumes normal operations. Recording allocations persistently allows them to be either re-discovered or released in a timely manner. If the log is irretrievably lost, then the workflow must be resumed from the beginning; previously acquired leases will eventually expire.

In contrast, the catalog server uses soft state. Since the catalog is only used to discover new resources, there is no need to recover old state from a crash. When the catalog is unavailable, the scheduler will continue to operate on known resources, but will not discover new ones. When the catalog server recovers, it rebuilds its knowledge as compute and storage servers send periodic updates.

The scheduler waits for passive indications of failure in compute and storage servers and then conducts active probes to verify. For example, if a job exits abnormally with an error indicating a failure detected by the interposition agent, then the scheduler suspects that the storage servers housing one or more of the volumes assigned to the job are faulty. The scheduler then probes the servers. If all volumes are healthy, it assumes the job encountered transient communication problems and simply reruns it. However, if the volumes have failed or are unreachable for some period of time, they are assumed lost.

The failure of a volume affects the jobs that use it. As a design simplification, the scheduler considers a partial volume failure to be a failure of the entire volume; in the future we plan to investigate the trade-offs involved in the choice of failure granularities. Running jobs that rely on a failed volume must be stopped. In addition, failures can cascade; completed processes that *wrote* to a volume must be rolled back and re-run. In order to avoid these expensive restarts of a pipeline, the scheduler may checkpoint scratch volumes as pipeline stages complete.

Of course, determining an optimal checkpoint interval is an old problem [27]. The solution depends upon the likelihood of failure, the value of a checkpoint, and the cost to create it. Unlike most systems, BAD-FS can solve this problem automatically, because the scheduler is in a unique position to measure the controlling variables. The scheduler performs a simple cost-benefit analysis at runtime to determine if a checkpoint is worthwhile.

The algorithm works as follows. The scheduler tracks the average time to replicate a scratch volume. This cost is initially assumed to be zero in order to trigger at least one replication and measurement. To determine the benefit of replication, the scheduler tracks the number of job and storage failures and computes the mean-time-to-failure across all devices in the system. The benefit of replicating a volume is the sum of the run times of those jobs completed so far in the applicable pipeline multiplied by the probability of failure. If the benefit exceeds the cost, then the scheduler replicates the volume on another storage server as insurance against failure. If the original fails, the scheduler restarts the pipeline using the saved copy.

Due to its robust failure semantics, the scheduler need not handle network partitions any differently than other failures. When partitions are formed between the scheduler and compute servers, the scheduler may choose to reschedule any jobs that were running on the other side of the partition. In such a situation, it is possible that the partition could be resolved, at which point the scheduler will find that multiple servers are executing the same jobs. Note that this will not introduce errors because each job writes to distinct scratch volumes. The scheduler may choose one output to extract and then discard the other.

3.4 Practical Issues

One of the primary obstacles to deploying a new distributed system is the need for a friendly administrator. Whether deploying an operating system, a file system, or a batch system, the vast majority of such software requires a privileged user to install and oversee the software. Such requirements make many forms of distributed computing a practical impossibility; the larger and more powerful the facility, the more difficult it is for an ordinary user to obtain administrative privileges. To this end, BAD-FS is packaged as a *virtual batch system* that can be deployed over an existing batch system without special privileges. This technique is patterned after the "glide-in job" described by Frey *et al.* [26] and is similar in spirit to recursive virtual machines [22].

To run BAD-FS, an ordinary user need only to be able to submit jobs into an existing batch system. BAD-FS bootstraps itself on these systems, relying on the basic ability to queue and run a self-extracting executable program containing the storage and compute servers and the interposition agent. Once deployed, the servers report to a catalog server, and the scheduler may then harness their resources. Note that the scheduling of the virtual batch jobs is at the discretion of the host system; these jobs may be interleaved in time and space with jobs submitted by other users. We have used this technique to deploy BAD-FS over several existing Condor and PBS batch systems.

Another practical issue is security. BAD-FS currently uses the Grid Security Infrastructure (GSI) [24], a public key system that delegates authority to remote processes through the use of time-limited proxy certificates. To bootstrap the system, the submitting user must enter a password to unlock the private key at his/her home node and generate a proxy certificate with a user-settable timeout. The proxy certificate is delegated to the remote system and used by the storage servers to authenticate back to the home storage server. This requires that users trust the host system not to steal their secrets, which is reasonable in a c2c environment.

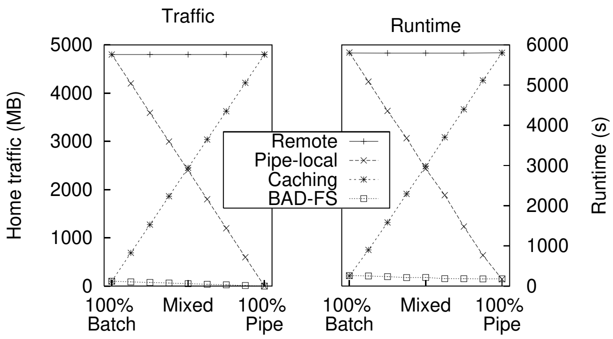


Figure 4: **I/O Scoping: Traffic Reduction and Run Times.** *These graphs show the total amount of network traffic generated by and runtimes for a number of different workloads with different optimizations enabled. For this experiment, we run 48 synthetic pipelines of depth 4, each of which generates a total of 100 MB I/O. Across the x-axis we vary the relative amounts of batch I/O and pipeline I/O. For example, at 100% Batch, the workload generates 100 MB of batch I/O and no pipeline. As is common in these types of workloads, the amount of endpoint I/O is small (1 KB). The leftmost graph shows the total amount of home server traffic; the right shows total runtimes when the home server is accessed over an emulated wide-area network (set at 1 MB/s).*

4 Experimental Evaluation

In this section, we present an experimental evaluation of BAD-FS under a variety of workloads. We first present our methodology, and then focus on I/O scoping, capacity-aware scheduling, and failure handling, using synthetic workloads to understand system behavior. Second, we present our experience with running real workloads on our system in a controlled environment. Finally, we discuss our initial experience using BAD-FS to run real workloads across multiple clusters in the wild.

4.1 Methodology

In the initial experiments in this section, we build an environment similar to that described in Section 2. We assume the user's input data is stored on a home server; once all pipelines have run and all output data is safely stored back at the home server, the workload is considered complete.

We assume that the workload is run on a remote cluster of machines, accessible from the user's home server via a wide-area link. To emulate this scenario, we limit the bandwidth to the home server to 1 MB/s via a simple network delay engine similar to DummyNet [44]. Thus, all I/O between the remotely run jobs and the home server must traverse this slow link. The cluster itself is taken from a dedicated compute pool of Condor nodes at the University of Wisconsin, connected via a 100 Mbit/s Ethernet switch. Each node has two Pentium-3 processors, 1 GB of physical memory and a 9 GB IBM SCSI drive, of which only a 1 GB partition is made available to Condor jobs. Of these 1 GB partitions, typically only about half is available at any one time as the rest awaits lazy garbage collection.

To explore the performance of BAD-FS under a range of workload scenarios, we utilize a parameterized synthetic batch-pipelined workload. The synthetic workload can be configured to perform varying amounts of endpoint, batch, and pipeline I/O, compute for different lengths of time, and can exhibit different amounts of both batch and pipeline parallelism. As each experiment requires different parameters, we leave those descriptions for the individual figure captions. However, given our previous results in workload analysis [54], we focus on *batch-intensive* workloads, which exhibit a high degree of batch sharing but little pipeline or endpoint I/O, and *pipe-intensive*, which perform large amounts of pipeline I/O but generate little batch or endpoint I/O.

4.2 I/O Scoping

The results of the first experiment, as shown in Figure 4, demonstrate how BAD-FS uses I/O scoping to minimize traffic across the wide area by localizing pipeline I/O in scratch volumes and reusing batch data in cooperative cache volumes. Although these optimizations are straightforward, their ability to increase throughput is significant.

In this experiment, we repeatedly run the same synthetic workload but vary the relative amount of batch and pipeline I/O. We compare a number of different system configurations. In the *remote* configuration, all I/O is sent to the home node. Against this baseline, we compare the *pipeline localization* and *caching* optimizations. Finally, both optimizations are combined in the BAD-FS configuration. Note that in these experiments, we assume copious cache space and a controlled environment; neither capacity-aware scheduling nor failure recovery is needed.

The left-hand graph shows the total I/O that is transferred over the wide-area network. Not surprisingly, the cooperative cache greatly reduces batch traffic to the home node by ensuring that all but the first reference to a batch data set is retrieved from the cache. We can also see that the pipeline localization optimizations work as expected, removing pipeline I/O entirely from the home server. Finally, we see that neither optimization in isolation is sufficient; only the BAD-FS configuration that combines both is able to minimize network traffic throughout the entire workload range. The right-hand graph in Figure 4 shows the runtimes of the workloads on our emulated remote cluster. From this graph, we can see the direct impact that wide-area traffic has on runtime.

4.3 Capacity-Aware Scheduling

Next, we examine the benefits of explicit storage management. The previous experiments were run in an environment where storage was not used to near capacity. With the increasing size of batch data sets and storage sharing by jobs and users, the scheduler must carefully manage remote space so as to avoid wide-area thrashing.

For these experiments, we compare the capacity-aware BAD-FS scheduler to two simple variants: a *depth-first* scheduler and a *breadth-first* scheduler. These algorithms are not aware of the data needs of the workload and

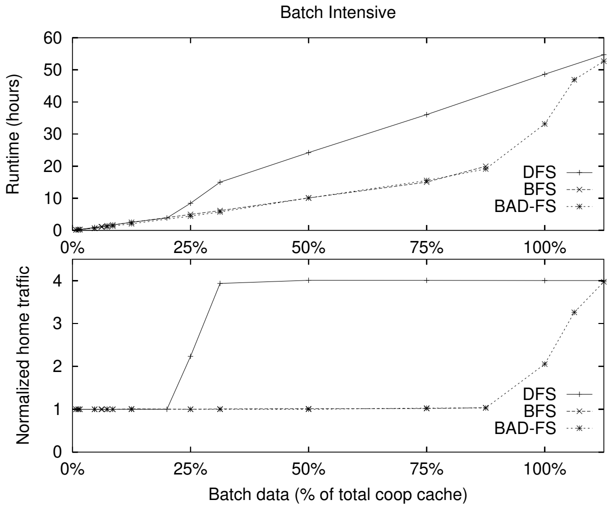


Figure 5: **Batch-intensive Explicit Storage Management.** *These graphs show the benefits of explicit storage management under a batch-intensive workload. The workload consists of 32 4-stage pipelines; within each stage, each process streams through a shared batch file (i.e., there are 4 batch files total). Batch file size is varied as a percentage of the total amount of cooperative cache space available across the 16 nodes in the experiment. All other I/O amounts are negligible. Each of 16 nodes has local storage which is used as a portion of the cache. The total cache size available is set to 8 GB (100% on the x-axis), which reflects our observations of available storage in the UW Condor pool.*

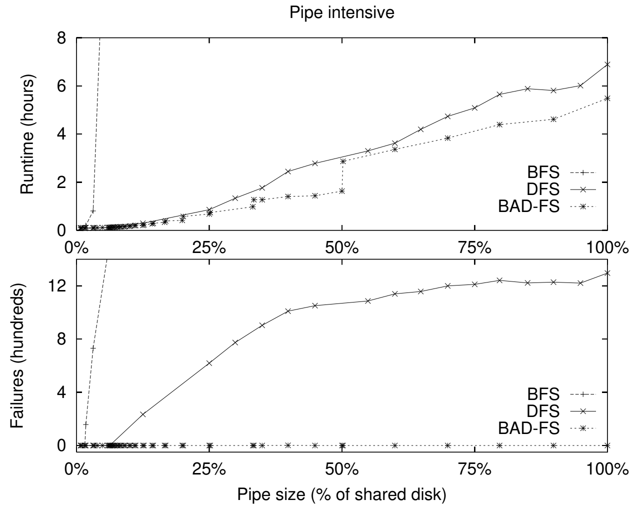


Figure 6: **Pipe-intensive Explicit Storage Management.** *These graphs depict the benefits of explicit storage management under a pipe-intensive workload. The workload consists of 32 4-stage pipelines, and pipe data size is varied as a percent of total storage available. All other I/O amounts are negligible. There are 16 compute servers and 1 storage server in this experiment (representing a set of diskless clients and a single server). The storage space at the server is constrained to 512 MB.*

base decisions solely on the job structure of its workflow. Depth-first simply assigns a single pipeline to each available CPU and runs all jobs in the pipeline to completion before starting another. Conversely, breadth-first attempts to execute all jobs in a batch to completion before descending to the next horizontal batch slice.

Each is correct for certain types of workloads, but can lead to poor storage allocations in others. For example, depth-first scheduling of a batch-intensive workload is more likely to cause thrashing because it attempts to simultaneously cache all of the batch datasets. Similarly, breadth-first scheduling of a pipe-intensive workload is more likely to over-allocate storage because it creates allocations for all pipelines before completing any.

4.4 Batch-intensive Capacity-Aware Scheduling

Figure 5 illustrates the importance of capacity-aware scheduling through measurements of batch-intensive workloads scheduled using various algorithms. Each workload is of depth four and thus has four large batch data sets, each of which takes up some sizable fraction of the available cooperative cache in the remote cluster (as varied along the x-axis). The upper graph shows the runtime and the lower presents the amount of wide-area traffic generated, normalized to the size of the batch data.

We can make a number of observations from these graphs. First, the similarity between the graphs validates that the wide-area network link is the bottleneck resource. Second, as expected, the different policies achieve similar results as long as the entirety of all four batch data sets fits within the caches (*i.e.*, up to 25%). As the size of the batch data approaches the total capacity of the cooperative cache, the runtime and wide-area traffic increase for depth-first scheduling. As the total batch data no longer fits in cache, depth-first scheduling must refetch batch data for each pipeline. In this case, this results in three extra fetches because with 64 pipelines and 16 compute servers, each server executes four pipelines.

Third, note that the runtime actually begins to increase slightly before 25%. The reason for this inefficiency is the lack of complete global control allowed through the current volume interface. In this case, the local cooperative cache hash function is not perfectly distributing data across its peers; when the cache nears full utilization, this skew overloads some nodes and results in extra traffic to the home server. Because we believe that this trade-off between local and global control is correct, the implication here is that the scheduler must be aware not only of the overall utilization of the cooperative cache, but also of the utilization of each peer.

Finally, breadth-first and BAD-FS scheduling are able to retain linear performance in this regime because they ensure that the total amount of batch data accessed at any one time does not exceed the capacity of the cooperative cache. However, once each individual batch dataset exceeds the capacity of the cooperative cache, the performance of breadth-first and BAD-FS scheduling converges with that of depth-first. Note that the same inefficiency that caused depth-first to deviate slightly before 25% causes this to happen slightly before 100%.

4.5 Pipe-intensive Capacity-Aware Scheduling

In our next set of cache management experiments, we focus on a pipeline-intensive workload instead of a batch-

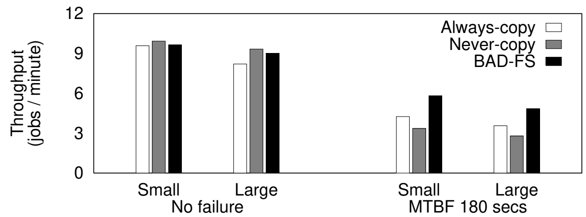


Figure 7: **Failure Handling.** *This graph shows the behavior of the cost-benefit strategy under different failure scenarios. Shown are two different workloads of width 64, depth 3 and one minute of CPU time; one performs a small amount of pipeline I/O, the other a large amount. Each is run both during periods of high and low rates of failure. Failures were induced using an artificial failure generator which formatted disks at random with a mean time between failures of 180 seconds, corresponding to the total runtime of a single pipe.*

intensive one. In this case, we expect the capacity-aware approach to follow the depth-first strategy more closely. Results are presented in Figure 6.

In the lower graph, we plot the number of failed jobs that each strategy induces. Job failure arise in this workload when there is a shortage of space for pipeline output; in such a scenario, a job that runs out of space for pipeline data aborts and must be rerun at some later time. Hence, the number of job failures due to lack of space is a good indicator of the scheduler's success in scheduling pipeline-intensive jobs under space constraints.

From the graph, we can observe that breadth-first scheduling is unable to prevent thrashing. In contrast, the capacity-aware BAD-FS scheduler does not exceed the available space for pipelines and thus never observes an aborted job. This careful allocation results in a drastically reduced runtime which is shown in the upper graph.

The stair-step pattern in the runtime of BAD-FS results from this careful allocation. When the size of the data in each pipeline is between 25% and 33% of the total storage, BAD-FS schedules workload jobs on only 3 of the 16 available CPUs; between 33% and 50% on just two; and as the data exceeds 50%, BAD-FS allocates only a single CPU at a time. Notice that BAD-FS achieves runtimes comparable or better than that of depth-first scheduling without any wasted resource consumption.

4.6 Failure Handling

We now show the behavior of BAD-FS under varying failure conditions. Recall that unlike traditional distributed systems, the BAD-FS scheduler knows exactly how to re-create a lost output file; therefore, whether to make a replica of a file on the remote cluster should depend on the cost of generating the data versus the cost of replicating it. This choice varies with the workload and the system conditions. Figure 7 shows how the BAD-FS cost-benefit analysis adapts to a variety of workloads and conditions. We compare to two naive algorithms: *always-copy*, which replicates a pipeline volume after each of its stages completes and *never-copy*, which does not replicate at all.

We draw several conclusions from this graph. In an environment without failure, replication leads to excessive overhead that increases with the amount of data. In this case BAD-FS outperforms always-copy but does not quite match never-copy because of the initial replication it needs to seed its analysis. In an environment with frequent failure, it is not surprising that BAD-FS outperforms never-copy. Less intuitively, BAD-FS also outperforms always-copy. In this case, given the particulars of the workload and the failure rate, replicating is only worthwhile after the second stage; BAD-FS correctly avoids replicating after the first stage while always-copy naively replicates after all stages.

4.7 Workload Experience

We conclude with demonstrations of the system running real workloads. In the first demonstration as presented in Figure 8, we compare the runtime performance of BAD-FS to other methods of utilizing local storage resources. In the *remote* configuration, local storage is not utilized at all and all I/O is executed directly at the home node. *Standalone* emulates AFS by caching data at the execute nodes but without any cooperative caching among their storage servers. The leftmost graph shows results for remote workload execution in which the bandwidth to the home server was constrained at 1 MB/s; the rightmost shows local workload execution in which the home server was situated within the same local area network as the execute nodes.

From these graphs, we can draw several conclusions. First, BAD-FS equals or exceeds the performance of remote I/O or standalone caching for all of the workloads in all of the configurations. These workloads, which are discussed in great detail in our earlier profiling work [54], all have large degrees of either batch or pipeline data sharing. Note that workloads whose I/O consists entirely of endpoint data would gain no benefit from our system.

Second, the benefit of caching, either cooperatively or in standalone mode, is greater for batch-intensive workloads, such as BLAST, than it is for more pipe-intensive ones such as HF. In these pipe-intensive workloads, the important optimization is pipeline localization, which is performed by both BAD-FS and standalone.

Third, cooperative caching in BAD-FS can outperform standalone both during cold and warm phases of execution. If the entire batch data set fits on each storage server, then cooperative caching is only an improvement while the data is being initially paged in. However, should the data exceed the capacity of any of the caches, then cooperative caching, unlike standalone, is able to aggregate the cache space and fit the working set.

This benefit of cooperative caching with warm caches is illustrated in the BLAST measurements in the graph on the left of Figure 8. Logfile analysis showed that

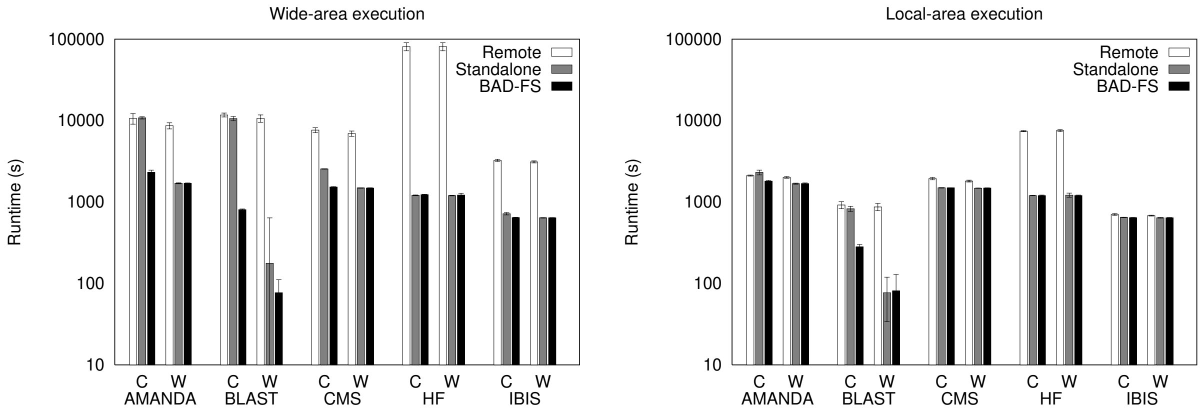


Figure 8: **Workload Experience.** *These graphs show runtime measurements of real workloads. For each workload, we submit 64 pipelines into a dedicated Condor pool of 16 CPUs. This Condor pool accesses local storage resources in one of three configurations:* remote *in which all I/O is redirected back to the home node;* standalone*, which emulates AFS-like caching to the home server; and* BAD-FS*. For each measurement, we present average runtime for the first jobs to run on each storage server when the storage cache is* cold *(C) and for the subsequent jobs which run when the cache is* warm *(W). The graph on the left shows runtimes when the workload is executed on a cluster separated from the home node by an emulated wide-area link (again set to 1 MB/s). On the right the home node is located within the same local area network. Note that the y-axis is shown in log scale to accentuate points of interest. Detailed information about these workloads can be found in our profiling study [54].*

two of the storage servers had slightly less cache space ($\approx$500 MB) than was needed for the total BLAST batch data ($\approx$600 MB). As subsequent jobs accessed these servers, they were forced to refetch data. Refetching it from the wide-area home server in the standalone case was much more expensive than refetching from the cooperative cache as in BAD-FS. With a local-area home server this performance advantage disappears. The different behavior of these two servers also explains the increased variability shown in these measurements.

Fourth, the penalty for performing remote I/O to the home node is less severe but still significant when the home node is in the same local-area network as the execute cluster. This result illustrates that BAD-FS can improve performance even when the bandwidth to the home server is not obviously a limiting resource.

Finally, comparing across graphs we make the further observation that BAD-FS performance is almost independent of the connection to the home server when caches are cold and becomes independent once they are warm. Using I/O scoping, BAD-FS is able to achieve local performance in remote environments.

4.8 In the Wild

Thus far, our evaluations have been conducted in controlled environments. We conclude our experimental presentation with a demonstration that BAD-FS is capable of operating in an uncontrolled, real world environment.

We created a wide-area BAD-FS system out of two existing batch systems. At the University of Wisconsin (UW), a large Condor system of over one thousand CPUs, including workstations, clusters, and classroom machines, is shared among a large number of users. At the University of New Mexico (UNM), a PBS system manages a cluster of over 200 dedicated machines.

We established a personal scheduler, catalog, and home storage server for our use at Wisconsin and then submitted a large number of BAD-FS bootstrap jobs to both batch systems without installing any special software at either of the locations. We then directed the scheduler to execute a large workload consisting of 2500 CMS pipelines using whatever resources became available.

Figure 9 is a timeline of the execution of this workload. As expected, the number of CPUs available to us varied widely, due to competition with other users, the availability of idle workstations (at UW), and the vagaries of each batch scheduler. UNM consistently provided twenty CPUs, later jumping to forty after nine hours. Two spikes in the available CPUs between 4 and 6 hours are due to the crash and recovery of the catalog server; this resulted in a loss of monitoring data, but not of running jobs.

The benefits of cooperative caching are underscored in such a dynamic environment. In the bottom graph, the cumulative read traffic from the home node is shown to have several hills and plateaus. The hills correspond to large spikes in the number of available CPUs.

Whenever CPUs from a new subnet begin executing, they fetch the batch data from the home node. However, smaller hills in the number of available CPUs do not have an effect on the amount of home read traffic because a new server entering an already established cooperative cache is able to fetch most of the batch data from its peers.

Finally, Figure 9 illustrates that both the design and implementation of BAD-FS are suitable for running I/O intensive, batch-pipelined workloads across multiple, uncontrolled real world clusters. Through failures and disconnections, BAD-FS continues making steady progress, removing the burden from the user of scheduling, monitoring, and resubmitting jobs.

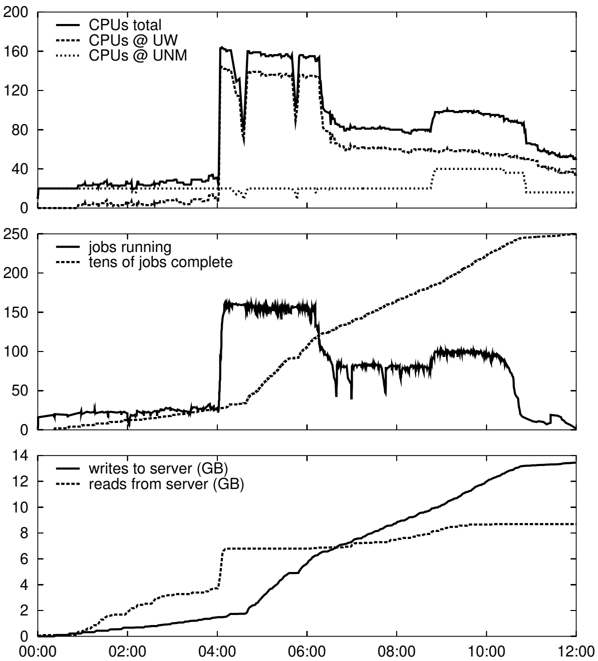


Figure 9: **In the Wild** *These three graphs present a timeline of the behavior of a large CMS workload run using BAD-FS. The workload consisted of 2500 CMS pipelines and was run wherever resources could be scavenged from a collection of CPUs at the University of New Mexico running PBS and from CPUs at the University of Wisconsin running Condor. The topmost timeline presents the total number of CPUs, the middle shows number of jobs running and cumulative jobs completed, and the bottom shows the cumulative traffic incurred at the home server.*

5 Related Work

In designing BAD-FS, we drew on related work from a number of distinct areas. Workflow management has historically been the concern of high-level business management problems involving multiple authorities and computer systems in large organizations, such as approval of loans by a bank or customer service actions by a phone company [28]. Our scheduler works at a lower semantic level than such systems; however, it does borrow several lessons from them, such as the integration of procedural and data elements [47]. The automatic management of dependencies for both performance and fault tolerance is found in a variety of tools [10].

Many other systems have also managed dependencies among jobs. A basic example is found with the UNIX tool `make`. More elaborate dependency tracking has been explored in Vahdat and Anderson's work on transparent result caching [56]; in that work, the authors build a tool that tracks process lineage and file dependency automatically. Our workflow description is a static encoding of such knowledge.

The manner in which the scheduler constructs private namespaces for running workloads is reminiscent of database views [32]. However, a private namespace is simpler to construct and maintain; views, in contrast, present systems with many implementation challenges, particularly when handling updates to base tables and their propagation into extant materialized views.

BAD-FS could be further improved through the prefetching of batch datasets. Other work [13] has noted the difficulty in correctly predicting future access patterns. In BAD-FS, however, these are explicitly supplied by the user via the declarative workflow description.

There has been much recent work in peer-to-peer storage systems [1, 4, 15, 35, 39, 46, 48]. Although each of these systems provides interesting solutions to the problem domain for which they are intended, each falls short when applied to the context of batch workloads, for the same reasons that distributed file systems are not a good match. However, many of the overlays developed for these environments may be useful for communication between clusters, something we plan to investigate in future work. Similar to p2p is work within grid computing [25], which uses many of the same techniques but is designed, as is BAD-FS, for c2c environments. One such example is Cluster-on-Demand [14] which offers sophisticated resource clustering techniques that could be used by BAD-FS to form cooperative cache groupings.

Extensible systems also share our approach of allowing the application more control [9, 19, 51]. Although recent work has recently revisited this approach [5], extensible systems have not been commercially successful because the need for specialized policies is not so great. We believe this need is greater for batch workloads running on systems designed for interactive use.

Some research in mobile computing bears similarity as well. Flinn *et al.* discuss the process of data staging on untrusted surrogates [21]. In many ways, such a surrogate is similar to the BAD-FS storage server; the major difference is that the surrogate is primarily concerned with trust, whereas our servers are primarily concerned with exposing control. Both Zap [40] and VMWare [49] allow for the checkpointing and migration of either processes or operating systems. We create a remote virtual environment, but at the much higher level of a batch system. Systems with secure interposition such as Janus [30] complement BAD-FS as they should make resource owners more willing to donate their resources into shared pools.

Finally, BAD-FS is similar to other distributed file systems. The Google File System [29] was also motivated by workloads that deviate from earlier file system assumptions. An additional similarity is a simplified consistency implementation; however, GFS must relax consistency semantics to enable this, while BAD-FS does so through explicit control. Earlier work on Coda, and AFS before it, is also applicable [34]. These systems use caching for availability, so as to allow disconnected operation. In BAD-FS, storage servers enact a similar role.

6 Conclusions

"He's a big bad wolf in your neighborhood;
not bad meaning bad but bad meaning good."
Run DMC, from 'Peter Piper'

Allowing external control has long been recognized as a powerful technique to improve many aspects of system performance. By moving control to the external user of a system, that system allows the user to dictate the policy that is most appropriate to the individual nature of their work. Systems lacking mechanisms for external control can only speculate. However, many systems have proven to be adept at speculation and work well for the majority of their workloads. In this paper we have argued that the distinct nature of batch workloads is not well matched by the design of traditional distributed file systems and the need therefore for external control is greater.

We have described BAD-FS, a distributed file system that exposes internal control decisions to an external scheduler. Using detailed knowledge of workload characteristics, the scheduler carefully manages remote resources and facilitates the execution of I/O intensive batch jobs on both wide-area and local-area clusters. Through synthetic and real workload measurements in both controlled and uncontrolled environments, we have demonstrated the ability of BAD-FS to use workload specific knowledge to improve throughput by selecting appropriate storage policies in regards to I/O scoping, space allocation and cost-benefit replication.

7 Acknowledgments

We would like to thank Nate Burnett, Nicholas Coleman, Tim Denehy, Barry Henrichs, Florentina Popovici, Muthian Sivathanu and Vijayan Prabhakaran from our department for their helpful discussions and comments on this paper. We are grateful for the excellent support provided by the members of the UW CSL.

We also wish to state our appreciation to Jeff Chase for his thoughtful analysis of our work throughout the development of this project. Finally, we thank our anonymous reviewers for their many helpful suggestions and Eric Brewer for his excellent and insightful shepherding, which has substantially improved both the content and the presentation of this paper.

This work is sponsored in part by NSF CCR-0092840, NSF UF00111, NGS-0103670, CCR-0133456, CCR-0098274, ITR-0086044, and ITR-0325267, DOE DE-FC02-01ER25450, the Wisconsin Alumni Research Foundation, EMC, and IBM.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec 2002.

[2] R. Agrawal, T. Imielinski, and A. Swami. Database Mining: A Performance Perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, Dec 1993.

[3] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, , and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. In *Nucleic Acids Research*, pages 3389–3402, 1997.

[4] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, and R. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 109–26, Copper Mountain, CO, Dec 1995.

[5] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. Nugent, and F. I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), NY, Oct 2003.

[6] Avery, P. et al. CMS Virtual Data Requirements. kholtman.home.cern.ch/kholtman/tmp/cmsreqsv6.ps, 2001.

[7] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, CA, Oct 1991.

[8] J. Bent, V. Venkataramani, N. Leroy, A. Roy, J. Stanley, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Flexibility, Manageability, and Performance in a Grid Storage Appliance. In *Proceedings of High-Performance Distributed Computing (HPDC-11)*, pages 3–12, Edinburgh, Scotland, Jul 2002.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 267–284, Copper Mountain, CO, Dec 1995.

[10] Y. Breitbart, A. Deacon, H.-J. Schek, A. P. Sheth, and G. Weikum. Merging Application-centric and Data-centric Approaches to Support Transaction-oriented Multi-system Workflows. *SIGMOD Record*, 22(3):23–30, 1993.

[11] J. F. Cantin and M. D. Hill. Cache Performance for Selected SPEC CPU2000 Benchmarks. Computer Architecture News (CAN), Sep 2001.

[12] S. Chandra, M. Dahlin, B. Richards, R. Y. Wang, T. E. Anderson, and J. R. Larus. Experience with a Language for Writing Coherence Protocols. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, Oct 1997.

[13] F. W. Chang and G. A. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 1–14, New Orleans, Louisiana, Feb 1999.

[14] J. S. Chase, L. E. Grit, D. E. Irwin, J. D. Moore, and S. Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC 12)*, Seattle, WA, June 2003.

[15] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, Oct 2001.

[16] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, Monterey, CA, Nov 1994.

[17] EDA Industry Working Group. The EDA Resource. http://www.eda.org/, 2003.

[18] D. A. Edwards and M. S. McKendry. Exploiting Read-Mostly Workloads in The FileNet File System. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*, pages 58–70, Litchfield Park, Arizona, Dec 1989.

[19] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain, CO, Dec 1995.

[20] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 201–212, Copper Mountain, CO, Dec 1995.

[21] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanarayanan. Data Staging on Untrusted Surrogates. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, CA, Apr 2003.

[22] B. Ford, M. Hibler, J. Lepreau, P. Tullman, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, Seattle, WA, Oct 1996.

[23] I. Foster and P. Avery. Petascale Virtual Data Grids for Data Intensive Science. GriPhyn White Paper, 2001.

[24] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[25] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[26] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A Computation Management Agent for Multi- Institutional Grids. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC 10)*, San Francisco, CA, Aug 2001.

[27] E. Gelenbe. On the Optimal Checkpoint Interval. *Journal of the ACM*, 26(2):259–270, Apr 1979.

[28] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[29] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), NY, Oct 2003.

[30] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the Sixth USENIX Security Symposium*, July 1996.

[31] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, CA, Oct 2000.

[32] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques and Applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, 1995.

[33] M. B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 80–93, Asheville, North Carolina, Dec 1993.

[34] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), Feb 1992.

[35] J. Kubiatowicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 190–201, Cambridge, MA, Nov 2000.

[36] T. L. Lancaster. The Renderman Web Site. http://www.renderman.org/, 2002.

[37] W. Litwin, M.-A. Neimat, and D. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB 20)*, pages 342–353, Santiago, Chile, Sep 1994.

[38] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of ACM Computer Network Performance Symposium*, pages 104–111, June 1988.

[39] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec 2002.

[40] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec 2002.

[41] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP '85)*, pages 15–24, Orcas Island, WA, Dec 1985.

[42] Platform Computing. Improving Business Capacity with Distributed Computing. www.platform.com/industry/financial/, 2003.

[43] R. Raman. *Matchmaking Frameworks for Distributed Resource Management*. PhD thesis, University of Wisconsin-Madison, Oct 2000.

[44] L. Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[45] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 41–54, San Diego, CA, June 2000.

[46] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, Oct 2001.

[47] M. Rusinkiewicz and A. P. Sheth. Specification and Execution of Transactional Workflows. In *Modern Database Systems: The Object Model, Interoperability, and Beyond.*, pages 592–620. 1995.

[48] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming Aggressive Replication in the Pangaea Wide-area File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec 2002.

[49] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec 2002.

[50] M. Satyanarayanan. A Study of File Sizes and Functional Lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*, pages 96–108, Pacific Grove, CA, Dec 1981.

[51] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 213–228, Seattle, WA, Oct 1996.

[52] S. Soderbergh. Mac, Lies, and Videotape. www.apple.com/hotnews/articles/2002/04/fullfrontal/, 2002.

[53] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, , and D. Anderson. A New Major SETI Project based on Project Serendip Data and 100,000 Personal Computers. In *Proceedings of the 5th International Conference on Bioastronomy*, 1997.

[54] D. Thain, J. Bent, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Pipeline and Batch Sharing in Grid Workloads. In *Proceedings of High-Performance Distributed Computing (HPDC-12)*, pages 152–161, Seattle, WA, June 2003.

[55] D. Thain and M. Livny. Parrot: Transparent User-Level Middleware for Data-Intensive Computing. In *Workshop on Adaptive Grid Middleware*, New Orleans, Louisiana, Sep 2003.

[56] A. Vahdat and T. E. Anderson. Transparent Result Caching. In *Proceedings of the USENIX Annual Technical Conference (USENIX '98)*, New Orleans, Louisiana, June 1998.

[57] W. Vogels. File System Usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 93–109, Kiawah Island Resort, South Carolina, Dec 1999.